Delta User Guide

Guillaume Doyen, Alain Ploix, Marc Lemercier and Rida Khatoun
Institut Charles Delaunay/ERA – FRE CNRS 2848
Université de Technologie de Troyes
12 rue Marie Curie – 10000 TROYES – France
Contact: `guillaume.doyen@utt.fr`

February 24, 2010– v1.0

Contents

1 Introduction

Large scale evaluation is a mandatory step for the validation of any peer-to-peer (P2P) application. Nonetheless, while most of dedicated tools rely on simulation, distributed experimentation tools are rare. Experimentation in real conditions is crucial since (1) some proposals cannot be simulated and (2) it is a mandatory step towards the production of a functional implementation. For the moment, experimentation tools are mainly ad hoc ones dedicated to a specific context. This situation is not satisfying because a lot of time is wasted for each experimental setup and results coming from different proposals cannot be compared. Delta [1] is a generic environment for the large-scale evaluation of any kind of distributed applications but especially for P2P ones. The main features of Delta are: (1) support of large-scale experiments, (2) complete independence from the evaluated application, (3) the repeatability of experiments and (4) the definition of P2P standard actions and measurements.

This guide is intended to show how to get started with Delta. Section 2 is the actual user guide. It uses a single example built around the *Point* class which simulates a distributed application. It first gives the general method one has to follow for each new experiment. Then it details each step. Section 3 presents use-case examples of Delta. Specificaly, it focuses on two java-based implementations of DHT that are FreePastry and OpenChord.

[1] http://delta.utt.fr

Finally, Section 4 gives full listings of elements that are referenced in the user guide.

2 User guide

2.1 Requirements

Entirely written in Java, DELTA supports both Windows and Linux operating systems. To use a compiled version of DELTA, Java[2] (JDK 1.6 or later) is required. To compile it, Ant[3] (version 1.6.5 or later) is required too.

2.2 General method

In order to use DELTA as experimentation support, one has to follow steps described below.

1. Defining the interface between DELTA and the evaluated application;
2. Implementing the interface;
3. Writing the experimentation scenario;
4. Configuring DELTA for the scenario execution;
5. Executing the experimentation.

In the following, we explain each of these steps with a simple example. We chose not too include measurements aspects in these steps but we explain how to perform measurements with DELTA in a dedicated section.

2.3 Context of our example

The example we use along this user guide is based on objects standing for points. Here, a *Point* represents the individual element of the distributed application we want to experiment. In a concrete experimenation case, it could a peer, an agent, or any other kind of distributed piece of code.

Listing 1 gives the source code of the *Point* class. A *Point* is defined through two coordinates and a label. For each of these attributes, getter and setter are defined. Tranformation methods, namely move and invert, perform operations on attributes and finally the distance method returns

[2]http://java.sun.com
[3]http://ant.apache.org

the distance of the *Point* from the origin. This code was designed to simple since this guide does not focus on the evaluated application but rather on its integration in DELTA.

Listing 1: The *Point* class

```java
package fr.utt.era.deltaappls.point;

public class Point {

    private Integer x, y;
    private String label;

    public Point(String label, Integer x, Integer y) {
        this.x = x;
        this.y = y;
        this.label = label;
    }

    public void invert() {
        Integer i = x;
        x = y;
        y = i;
    }

    public void move(Integer dx, Integer dy) {
        x += dx;
        y += dy;
    }

    public double distance() {
        double d = Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2));
        return d;
    }

    public String toString() {
        return "Point (" + label + "," +
                       x.intValue() + "," +
                       y.intValue() + ")";
    }

    public Integer getX() {
        return x;
    }

    public void setX(Integer x) {
        this.x = x;
    }

    public Integer getY() {
        return y;
    }

    public void setY(Integer y) {
        this.y = y;
    }
    public String getLabel() {
        return label;
    }
}
```

2.4 Interfacing Delta with the evaluated application

2.4.1 Defining the interface

Given a distributed application, one need to define the actions that DELTA will perform on it. These actions are defined in an interface which must inherits from the DeltaRemoteInstance one. As shown in Listing 2, the latter defines a single method called `getMeasurement`. This method is the only one which is hardcoded in DELTA. All other method calls are done through introspection. Details on the `getMeasurement` method are given in section 2.8.

Listing 2: The DeltaRemoteInstance interface

```
package fr.utt.era.delta;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface DeltaRemoteInstance extends Remote {
    public DeltaMeasurement getMeasurement(
        String metricName,
        String measurementId,
        String component,
        String correlator
            ) throws RemoteException;
}
```

Thus, to startup with our example, Listing 3 proposes an interface that enable the control of one or several points through DELTA. It is to notice that the design of this interface is only lead by experimentation choices. Nonetheless, since during an experiment all methods calls are performed through RMI[4], each method must throw the `java.rmi.RemoteException`.

Listing 3: The PointRemoteInstance interface

```
package fr.utt.era.deltaappls.point;

import fr.utt.era.delta.DeltaRemoteInstance;
import java.rmi.RemoteException;

public interface PointRemoteInstance extends DeltaRemoteInstance {
```

[4]Remote Method Invocation

```
    public void createPoint(String label, Integer x, Integer y)
        throws RemoteException;
    public void movePoints(Integer dx, Integer dy)
        throws RemoteException;
    public void invertPoints()
        throws RemoteException;
}
```

2.4.2 Implementing the interface

Once the interface is defined, the next step consists in implementing it in what we call a server. We use this term because this class will create, host and let be reachable the application (here a point). Thus the class `PointServer` contains a collection of *Points*, here a vector and performs operations on it through the methods defined in the `PointRemoteInstance` instance. As for previous section, we do not give here explanation concerning the `getMeasurement` method which will be detailed later in this document.

Listing 4: The PointServer class

```
package fr.utt.era.deltaappls.point;

import fr.utt.era.delta.DeltaMeasurement;

import java.rmi.RemoteException;
import java.util.*;

public class PointServer implements PointRemoteInstance {

    private Vector<Point> points;

    public PointServer() {
        points = new Vector<Point>();
    }

    public void createPoint(String label, Integer x, Integer y)
            throws RemoteException {
        points.addElement(new Point(label, x, y));
    }

    public void movePoints(Integer dx, Integer dy)
            throws RemoteException {
        for (Enumeration e = points.elements(); e.hasMoreElements();) {
            ((Point) e.nextElement()).move(dx, dy);
        }
    }

    public void invertPoints() throws RemoteException {
        for (Enumeration e = points.elements(); e.hasMoreElements();) {
            ((Point) e.nextElement()).invert();
        }
    }
    public DeltaMeasurement getMeasurement (String metricName,
            String measurementId, String component, String correlator)
            throws RemoteException {
        // explained in section 2.8
        return null;
    }
}
```

2.5 Writing the experimentation scenario

In this section, we present the way an experimentation scenario can be built. Since all syntaxic elements of a scenario are not illustrated here, we give the DTD a scenario XML document has to follow in Listing 20 of Section 4.2.

2.5.1 Overview of a scenario execution

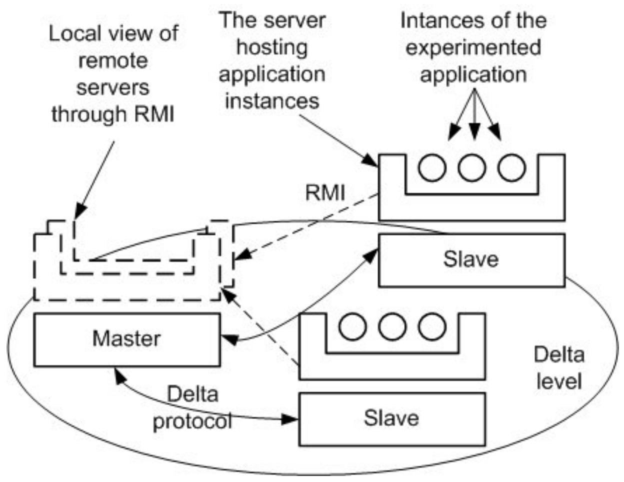
Local view of remote servers through RMI
The server hosting application instances
Intances of the experimented application
RMI
Slave
Master
Delta protocol
Slave
Delta level

Figure 1: Delta architecture

The scenario used in an experimentation is defined in a XML document which specifies what has to be done by each of Delta instances created. To clearly understand how a scenario is defined and executed, one need to understand the Delta architecture as depicted on Figure 1. At runtime, each Delta instance is executed in a different JVM[5]. Each instance plays a role: one is the master while others are slaves. The master is in charge of solely

[5]Java Virtual Machine

executing the scenario while slaves instantiate the application server and thus actually host the evaluated application. From a scenario perspective, this architecture has the following consequence: the master never appears in a scenario and targets are the slave instances.

Following is the description of each point a scenario has to define. It follows our *Point* example. The full version of the scenario can be found in Listing 19 of section 4.1.

2.5.2 Defining the scenario metadata

A scenario is composed of three parts. The first part gives metadata concerning the scenario (scenario author, scenario name, a description, a date, a version, . . .). This information is used to generate the header of results file easing thus the correlation between a scenario and the obtained results. An example of scenario metadata is given in Listing 5.

Listing 5: Metadata of a scenario

```
<metadata>
  <title>Points: scenario 1</title>
  <version>version 1 : 12/10/2007</version>
  <authors>Marc LEMERCIER</authors>
  <description>Affiche Points</description>
</metadata>
```

2.5.3 Defining the scenario topology

The second part of a scenario is the definition of the experiment topology. The latter is used to let the master knwo the number of expected slaves in the experiment and for each slave it gives a symbolic name used in the tasks and optionnaly associate it with a specific hostname. To illustrate this definition, consider the topology represented in Listing 6 executed by hosts named `slave1.utt.fr`, `slave2.utt.fr` and `slave3.utt.fr`. Since slave `slave1` is explicitely linked the host named `slave1.utt.fr` in the topology definition, actions targeted to `slave1` will be executed by `slave1.utt.fr`. Now, concerning `slave2` and `slave3` depending of the registration order, they will be indifferently attached to `slave2.utt.fr` or to `slave3.utt.fr`.

Listing 6: Topology of a scenario

```
<topology>
  <slave name='slave1' hostname='slave1.utt.fr'/>
  <slave name='slave2'/>
  <slave name='slave3'/>
</topology>
```

2.5.4 Defining the scenario tasks

The third part is a set of tasks to be executed. Each task is defined through three parameters that are: the action performed, the target slaves and the time constraints. Listing 7 gives an example of action that illustrates these settings.

Listing 7: Example of a task

```
<task ID='1' name='CreatePoints'>
  <time>
    <start>1</start>
    <stop>2</stop>
    <occurrence>2</occurrence>
  </time>
  <targets select='list'>
    <slave name='slave1'/>
    <slave name='slave2'/>
  </targets>
  <action mode='random'>
    <method-name>createPoint</method-name>
    <method-params>
      <param>
        <param-type>java.lang.String</param-type>
        <param-name>label</param-name>
        <param-value>point</param-value>
      </param>
      <param>
        <param-type>java.lang.Integer</param-type>
        <param-name>x</param-name>
        <param-value>10</param-value>
      </param>
      <param>
        <param-type>java.lang.Integer</param-type>
        <param-name>y</param-name>
        <param-value>20</param-value>
      </param>
    </method-params>
  </action>
</task>
```

Now we detail the mean as well as the different ways each parameter can be used.

Time: the time constraints are defined through a start time and stop time, given in seconds and a number of occurences, which is in other words the number of times task will be performed. The time slicing is done with the following manner. Let i be an occurence indice with $0 \leq i < occurence$, then $execution\_time_i = start + \frac{i*(stop-start)}{occurence}$.

Targets: are set through the `select` attribute which is either set to `list` or `topology`. In the first case, the list of targets is given as a child of this tag (e.g. Listing 7). In the second case, the whole topology is concerned by the action.

Action: represents the full definition of the method call, with the method name and for each method parameter, its name, its type and its value. The `mode` attribute tells which targets of the task will execute the action. Set to `random`, for each occurrence, only one randlomly chosen target will perform the action. Set to `all`, all the task targets will perform it.

N.B. : The value parameter of a method call has sometime to be the hostname of a slave. Since such a name cannot be *a priori* determined due to the dynamic bind of slaves names to hostnames, an escaping sequence enables DELTA to dynamically replace a slave name by the hostname, at runtime. This escape sequence is bound with '{' and '}' caracters. For example, topology given in Listing 6, `slave1` can be dynamically replaced by `slave1.utt.fr`, at runtime by writing {`slave1`} as a parameter value. This functionnality is especially interesting in P2P experiments for example to set bootstrap addresses.

2.5.5 Checking the scenario syntax

At runtime, when DELTA loads the experiment scenario, it checks the scenario syntax as defined in the DTD given in Listing 20 of Section 4.2. Thus a file named `scenario.dtd`, containing the scenario DTD must exist in the folder of the executed scenario. Otherwise, DELTA will stop and print a related error message. Future versions of DELTA will probably remove this limitation but for the moment one must deal with this constraint in order to let DELTA working properly.

2.6 Configuring Delta for the scenario execution

In order to execute your scenario properly, DELTA must be configured through a configuration file. At runtime, this file is read by both the master and slaves instances and contains parameters concerning the master instance, slaves instances and both of them. Following is the description of all these parameters.

2.6.1 Parameters common to both the master and the slaves

Following parameters are used by both the master and slave instances.

- `MASTER_IP`: gives the IP address of the Delta master instance. In case of a single host experiment, `127.0.0.1` is supported.
- `IS_SLAVE`: forces a Delta instance to run as a slave one and this even if the host on which it is executed owns the IP address identified as `MASTER_IP`. It is used for single host experiments for which only one of the several Delta instances has to behave as a master. See Section 2.6.4 for more information.
- `MASTER_PORT`: gives the TCP port number a master instance will listen on.
- `SLAVE_PORT`: gives the port a slave instance will use to connect to the master. If omitted, a random available port is chosen.
- `RMI_REGISTRY_PORT`: gives the port number the RMI registry will listen on. Note that each slave instance executes a RMI registry.
- `EXPERIMENT_ID`: an identifier that the master and its slaves exchanged to ensure that they are involved in the same experiment.

2.6.2 Master parameters

Following parameters are used by the master instance.

- `SCENARIO_FILE_NAME`: The path and name of the file containing the scenario the master instance will execute.
- `OUTPUT_FILE_NAME`: The path and name of the file that will contain results provided by the `getMeasurement` method. Note that (1) the specified path must exists before Delta is executed and (2) if a file already exists it will not be overwritten; new data will be appended at the end of it.
- `TRACE_FILE_NAME`: The path and name of the file that will contain the trace of all actions the master has executed during the experiment. This file is used for log and debug of scenarios. As for the previous parameter, note that (1) the specified path must exists before Delta is executed and (2) if a file already exists it will not be overwritten; new data will be appended at the end of it.
- `ACTIONS_SEQUENCEMENT`: This parameter can take two values `serial` and `paralell`. In the first case, all actions directed to more than one

target will be executed sequencially, with blocking calls. In the second case, all actions are executed concurrently.

2.6.3 Slave parameters

Following parameters are used by slave instances.

- `MAX_CONNECTION_ATTEMPTS`: Since no distinction has to be done between the master and slave instances during the deployment and execution phases of the code, slaves instances may try to connect to the master instance before it is started an ready to listen for slaves registration. This parameter, as well as the following one, address this possibility. Indeed, it sets the number of times a slave has to try connecting to the master instance before giving up and exiting.

- `CONNECTION_ATTEMPTS_TIMER`: This parameter, must be used with the previous one. It sets the number of seconds a slave waits between two connections attempts to the master instance.

- `REMOTE_DELTA_CLASS`: This parameter gives the name of the class that act as an application server, as described in Section 2.4.2. Note that since this class is instanciated through Delta, its constructor must not receive any parameter.

2.6.4 Running the master and the slaves on the same host

In case one wants to run both a master and a slave instance on the same host, two different configuration files have to be provided to Delta. The only difference between these files is the add of the `IS_SLAVE` parameter in the slave configuration file.

2.6.5 A configuration example

Given the different parameters described above, we give in Listing 8 an example of configuration file that can be used with the topology given in Listing 6. We assume here that the master has `192.168.0.1` as an IP address.

Listing 8: An example of Delta configuration

```
MASTER_IP=192.168.0.1
MASTER_PORT=9400
RMI_REGISTRY_PORT=1099
EXPERIMENT_ID=0123456789ABCDEF
```

```
SCENARIO_FILE_NAME=scenario.xml
OUTPUT_FILE_NAME=output.txt
TRACE_FILE_NAME=trace.txt
ACTIONS_SEQUENCEMENT=parallel

MAX_CONNECTION_ATTEMPTS=3
CONNECTION_ATTEMPTS_TIMER=10
REMOTE_DELTA_CLASS=PointServer
```

2.7 Executing the experimentation

2.7.1 Running Delta from the command line interface

When executing an experiment with DELTA one must not start `java` with the application server class or any other one. Indeed, DELTA wraps this creation. Thus, the class name that has to be passed to `java` is the DELTA class, namely `fr.utt.era.delta.Delta`. This class can be parametered as follows:

`-f [config-file]` : load the mentionned configuration file and start Delta.

`-w` : print warranty information and exit.

`-c` : print license conditions and exit.

Of course, all classes that have to be loaded by DELTA must be given in the classpath. DELTA requires two jar files that are `jdom.jar` and `log4j-1.2.9.jar`. Thus an execution of DELTA with an application that requires two jars (`appl1.jar` and `appl2.jar`) could look like that:

```
java -cp\
jdom.jar:log4j-1.2.9.jar:delta-1.2.jar:appl1.jar:appl2.jar\
fr.utt.era.delta.Delta -f delta.cfg
```

2.7.2 Traces of scenario events

In order to follow the execution of a scenario and check that it runs as expected, DELTA generates a trace file that logs all events, related to the executed scenario that occur. Three kind of events are written in this file:

Scenario: Start and stop times are logged.

Tasks: Start and stop times are logged.

Actions: For each occurrence of an action, a log is written. It follows this convention:

- Actual execution time in milliseconds;
- Expected execution time in milliseconds (as specified in the scenario);
- Task name;
- Method name;
- Target;
- Execution duration in microseconds.

Listing 9 is an example of trace file for the execution of the task depicted in Listing 7.

Listing 9: An example of scenario trace

```
==========================================================
TRACE FOR SCENARIO Points : scenario 1
----------------------------------------------------------
Title: Points : scenario 1
Author(s): Marc LEMERCIER
Execution date: Thu Apr 24 18:44:13 CEST 2008
----------------------------------------------------------

Heap memory: init = 0(0K) used = 701240(684K) committed = 5177344 ...
Non heap memory: init = 33751040(32960K) used = 14738144(14392K) ...
----------------------------------------------------------

0       Scenario 'Points : scenario 1' started.
1000    Task 'CreatePoints' started.
1004    1000    CreatePoints    createPoint     slave2  1621523 micros
1503    1500    CreatePoints    createPoint     slave1  4166597 micros
1997    Task 'CreatePoints' ended.
2005    Scenario 'Points : scenario 1' ended.
```

2.7.3 The Delta internal logger

To monitor the way Delta operates, a logger is defined for each Delta component. This logger is configured through a file whose name is hardcoded in Delta. Thus at runtime, in the root folder of your experiment, the file `logger.cfg` must exist as well as a `log` folder in which Delta will write an internal log file. Future versions of Delta will probably remove this limitation but for the moment one must deal with this constraint in order to let Delta working properly.

2.8 Performing measurements

2.8.1 The `getMeasurement` method

Up to this point, we saw how to use DELTA in order to run user-defined methods that take part of a experimentation scenario. Nonetheless, we have kept away the one which is the most important: `getMeasurement`. This method allows a user to perform any kind of measurements in its application and to collect them in the output file, as defined in the DELTA configuration file, on the master instance. The `getMeasurement` method is the only one whose definition is hard-coded in DELTA (cf. Listing 2). Nonetheless, we defined it to be generic enough so that can be used in any context. Parameters of the method are:

`metricName`: A textual string which represents the name of the metric. For example, in the application case of this guide, it could be `DISTANCE`.

`measurementId`: A string used to identify a measurement independantly from its name. This allow the user to correlate results coming from different hosts but taking part of the same measurement.

`component`: A string that identify the target component of the application. This parameter is usefull because in an application the same metric name may have to be used on different parts of an application.

`correlator`: A user-defined string that is passed to the application. It is intended to be used when measurements are themselves distributed and thus need to be correlated before being collected in the output file.

Now concerning the return of the method, Listing 2 shows that it is an object of class `DeltaMeasurement` whose code is given in Listing 10. This class is a simple abstract container for any type Java collection. It is designed in this manner because the implementation of this class will change according to the actual kind of measurements.

Listing 10: The DeltaMeasurement class

```java
package fr.utt.era.delta;

import java.util.Collection;
import java.io.Serializable;

public abstract class DeltaMeasurement implements Serializable {
    protected Collection results;
}
```

Thus, in order to collect measurements, one need to refine `DeltaMeasurement` and implement a concrete container of measurements. Back to our example, we defined a `PointMeasurement` class that acts as a container for several points measurements. A single measurement is coded through the `PointMeasurementUnit` private class.

Listing 11: The PointMeasurement class

```java
package fr.utt.era.deltaappls.point;

import java.io.Serializable;
import java.util.Vector;

import fr.utt.era.delta.DeltaMeasurement;

public class PointMeasurement extends DeltaMeasurement
        implements Serializable {

    private static final long serialVersionUID = -7306651319040545798L;

    public PointMeasurement () {
        results = new Vector<PointMeasurementUnit>();
    }

    public void addMeasurementUnit (String label, String v) {
        results.add(new PointMeasurementUnit(label, v));
    }

    public String toString() {
        return results.toString();
    }

    private class PointMeasurementUnit implements Serializable {
        private static final long
                serialVersionUID = -6931047266354749278L;
        private String name;
        private String value;
        public PointMeasurementUnit(String n, String v) {
            name = n;
            value = v;
        }
        public String toString() {
            return new String(name + "\t" + value);
        }
    }
}
```

2.8.2 Integrating measurements in your code and scenario

As for any method DELTA calls in a scenario, the body of the `getMeasurement` method has to be written by the user. Listing 12 gives an example of implementation that for each point on the host, collects its distance from the origin. Note that in this case we do not specify a target component nor a correlator.

Listing 12: An implementation example of the getMeasurement method

```java
public DeltaMeasurement getMeasurement (String metricName,
        String measurementId, String component, String correlator)
        throws RemoteException {
    PointMeasurement pm = new PointMeasurement();

    if (metricName.equals("DISTANCE")) {
        for (Enumeration<Point> e = points.elements();
                e.hasMoreElements();) {
            Point p = e.nextElement();
            String distance = Double.toString(p.distance());
            pm.addMeasurementUnit(p.getLabel(), distance);
        }
    } else {
        System.err.println("Unknown metric: " + metricName);
    }
    return pointMeasurement;
}
```

Listing 13: Performing measurements in a scenario

```xml
<task ID='2' name='measurement'>
  <time>
    <start>5</start>
    <stop>9</stop>
    <occurrence>2</occurrence>
  </time>
  <targets select='topology'/>
  <action mode='all'>
    <method-name>getMeasurement</method-name>
    <method-params>
      <param>
        <param-type>java.lang.String</param-type>
        <param-name>metricName</param-name>
        <param-value>DISTANCE</param-value>
      </param>
      <param>
        <param-type>java.lang.String</param-type>
        <param-name>measurementId</param-name>
        <param-value>0x0001</param-value>
      </param>
      <param>
        <param-type>java.lang.String</param-type>
        <param-name>component</param-name>
        <param-value>null</param-value>
      </param>
      <param>
        <param-type>java.lang.String</param-type>
        <param-name>correlator</param-name>
        <param-value>null</param-value>
      </param>
    </method-params>
  </action>
</task>
```

2.8.3 Gathering results of measurements

The results of measurements are written in the output file, as specified in the configuration file. In this file, each measurement is written on a single line that contains the following fields:

- Expected measurement time in milliseconds;
- Time of actual measurement start in milliseconds;
- Time of actual measurement end in milliseconds;
- Metric identifier;
- Occurrence number of the measurement;
- Target slave name;
- Component name;
- Metric name;
- Measurement value, as specified in the `toString methods` of the measurement class and measurement unit class.

Following is an example of output file for the measurement task described in Listing 13. We assume that this task is integrated in the same scenario that contains the `createPoints` task too (cf. Listing 7). Thus, `slave1` has no *Point* instance and returns an empty result when performing a measurement.

Listing 14: An example of output file

```
==========================================================
OUTPUT FOR SCENARIO Points : scenario 1
----------------------------------------------------------
Title: Points : scenario 1
Author(s): Marc LEMERCIER
Execution date: Wed Jul 16 11:40:26 CEST 2008
----------------------------------------------------------

5000 5023 5055 0x0001 1 slave1 null DISTANCE []
5000 5023 5055 0x0002 1 slave2 null DISTANCE [point 22,36 ...]
5000 5023 5055 0x0002 1 slave3 null DISTANCE [point 22,36 ...]
7000 7021 7022 0x0002 2 slave1 null DISTANCE []
7000 7021 7022 0x0002 2 slave2 null DISTANCE [point 22,36 ...]
7000 7021 7022 0x0002 2 slave3 null DISTANCE [point 22,36 ...]
```

3 Use case examples with existing P2P frameworks

In this section we give use examples of Delta with existing frameworks, namely FreePastry and OpenChord, two Java-based implementations of respectively the Pastry [1] and Chord [2] DHT. Before presenting these use, we present a generic interface we designed in order to refine Delta in the context of P2P applications.

3.1 The P2PRemoteInstance interface

In order to capture all the aspects specific to a P2P application, we designed the `P2PRemoteInstance` interface which inherits from the `DeltaRemoteInstance` one. As shown in Listing 15, this interface addresses two aspects of P2P application that are important to control in an experiment: the overlay creation and the nodes life-cycle management. The overlay creation is controled through the `createNode` and `setBootstrapAddress` methods whose name clearly indicate their role. The nodes life-cycle management is operated through the four last methods: `setAlive`, `setIdle`, `kill` and `killAll`. These methods are useful to control to control, for example, churn in a DHT. Nonetheless, as for any `RemoteInstance` used by Delta, the body of these methods has to be implemented by the user.

Listing 15: The P2PRemoteInterface

```
package fr.utt.era.delta.p2p;

import java.rmi.RemoteException;
import fr.utt.era.delta.DeltaRemoteInstance;

public interface P2PRemoteInstance extends DeltaRemoteInstance {
    public void createNode() throws RemoteException;
    public void setBootstrapAddress(String bootstrapAddress)
            throws RemoteException;

    public void setAlive() throws RemoteException;
    public void setIdle(Integer numberOfNodes) throws RemoteException;
    public void kill(Integer numberOfNodes) throws RemoteException;
    public void killAll() throws RemoteException;
}
```

Together with the `P2PRemoteInstance` interface, we designed and implemented the `P2PMeasurement` class too. This class defines a dedicated container for P2P measurements that integrates the state of a node (alive, idle or dead) according to life-cycle we defined. The code of this class is given in Listing 16.

Listing 16: The P2PMeasurement class

```
package fr.utt.era.delta.p2p;

import java.util.Vector;
import java.io.Serializable;

import fr.utt.era.delta.DeltaMeasurement;

public class P2PMeasurement extends DeltaMeasurement
        implements Serializable {

    private static final long serialVersionUID = -83027390510991422L;

    public P2PMeasurement () {
        results = new Vector<P2PMeasurementUnit>();
    }

    public void addMeasurementUnit (String nId, String s, String v) {
        results.add(new P2PMeasurementUnit(nId, s, v));
    }

    public String toString() {
        return results.toString();
    }

    private class P2PMeasurementUnit implements Serializable {
        private static final long
                serialVersionUID = -2219465867290080731L;
        private String nodeId;
        private String state;
        private String value;
        public P2PMeasurementUnit(String nId, String s, String v) {
            nodeId = nId;
            state = s;
            value = v;
        }
        public String toString() {
            return new String(nodeId + "\t" + state + "\t" + value);
        }
    }
}
```

3.2 FreePastry

The original goal which led us to design and implement DELTA is the lack of a such a tool in FreePastry[6], the Java implementation of the Pastry DHT [1]. Althrough FreePastry provides a simulator, from our knowledge, it is limited to a single JVM. Thus, our Pastry server class reuses a lot of code elements coming from FreePastry simulator classes but it enables the control of more that one simulator remotely.

[6]http://freepastry.rice.edu

From a Java perspective, the PastryServer class implements the P2P-RemoteInterface. Listing 17 gives the full code of it, showing thus how we integrated the simulator with Delta. The implementation of this class should be considered as an example but not as a reference since althrough it enables us to perform our tests appropriatly, it may have to be reconsidered in order to be used in another context.

Concerning the measurements, we illustrate here the need for the identification of components: for a node we measure the number of unique entries, named UNIQUE_ENTRIES, of both the leafset (LEAF_SET component) and the routing table (ROUTING_TABLE component).

Listing 17: The PastryServer class

```
package fr.utt.era.deltaappls.pastry;

import java.net.*;
import java.rmi.RemoteException;
import java.util.*;
import java.io.*;

import rice.pastry.*;
import rice.pastry.socket.SocketPastryNodeFactory;
import rice.pastry.standard.RandomNodeIdFactory;
import rice.pastry.dist.DistPastryNodeFactory;
import rice.environment.Environment;

import fr.utt.era.delta.*;
import fr.utt.era.delta.p2p.*;

public class PastryServer implements P2PRemoteInstance {
    private final static int MAX_NUMBER_OF_NODES = 15;

    private static int bindport = 5009;
    private static String bootaddr = null;
    private static int bootport = 5009;

    private PastryNodeFactory factory;
    private NodeIdFactory nidFactory;

    private int startedNodes;
    private Vector<PastryNode> aliveNodes;
    private Vector<PastryNode> idleNodes;
    private Vector<NodeStateController> idleThreads;

    private Random random;
    private Date date;

    private Environment env;

    public PastryServer() {
        env = new Environment();
        date = new Date();
        random = new Random(date.getTime());
        nidFactory = new RandomNodeIdFactory(env);

        try {
            factory = new SocketPastryNodeFactory(
                    nidFactory, bindport, env);
        }
        catch (IOException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }

        aliveNodes = new Vector<PastryNode>();
        idleNodes = new Vector<PastryNode>();
        idleThreads = new Vector<NodeStateController>();

        startedNodes = 0;
    }

    protected NodeHandle getBootstrap() {
        InetSocketAddress addr = null;
        addr = new InetSocketAddress(bootaddr, bootport);
        NodeHandle bshandle = ((DistPastryNodeFactory) factory).
                getNodeHandle(addr);
        return bshandle;
    }

    private PastryNode makePastryNode() {
        NodeHandle bootstrap = getBootstrap();
        PastryNode pn = factory.newNode(bootstrap);

        synchronized (pn) {
            while (!pn.isReady()) {
                try {
                    pn.wait();
                }
                catch (InterruptedException e) {
                    System.out.println(e);
                }
            }
        }
        return pn;
    }

    public void createNode() throws RemoteException {
        if (startedNodes < PastryServer.MAX_NUMBER_OF_NODES) {
            System.out.println("Starting node number " +
                    (startedNodes + 1));
            PastryNode pn = makePastryNode();
            aliveNodes.add(pn);
            startedNodes++;
        }
        else {
            System.err.println("Maximum number of nodes (" +
                    PastryServer.MAX_NUMBER_OF_NODES +
                    ") is reached. Cannot create a new one.");
        }
    }

    public void setBootstrapAddress(String bootstrapAddress)
            throws RemoteException {
        if (bootstrapAddress.startsWith("localhost")) {
            try {
                bootstrapAddress =
                        InetAddress.getLocalHost().getHostName() +
                        bootstrapAddress.substring(
                        bootstrapAddress.indexOf(":"));
            }
            catch (UnknownHostException e) {
                System.err.println("Cannot convert 'localhost'
                        to the actual host name.");
                System.exit(1);
            }
        }

        bootaddr = bootstrapAddress.substring(0,
                bootstrapAddress.indexOf(":"));
        bootport = Integer.valueOf(bootstrapAddress.substring(
                bootstrapAddress.indexOf(":") + 1)).intValue();
        bindport = Integer.valueOf(bootstrapAddress.substring(
                bootstrapAddress.indexOf(":") + 1)).intValue();
        System.out.println("Setting bootstrap address to " +
                bootaddr + "(port: " + bootport + ")");
    }

    public void kill(Integer numberOfNodes) throws RemoteException {
        if (numberOfNodes <= startedNodes) {
            for (int i = 1; i <= numberOfNodes.intValue(); i++) {
                int rank = random.nextInt(aliveNodes.size());
                PastryNode pn = aliveNodes.get(rank);
                pn.destroy();
                aliveNodes.remove(rank);
            }
        }
        else {
            System.err.println("Cannot kill " +
                    numberOfNodes +
                    " nodes (number of started nodes = " +
                    startedNodes + ").");
        }
    }

    public void setAlive() throws RemoteException {
        for (int i = 0; i < idleThreads.size(); i++) {
            NodeStateController sc = idleThreads.get(i);
            sc.setAlive();
        }
        aliveNodes.addAll(idleNodes);
        idleNodes.clear();
        idleThreads.clear();
    }
```

```
public void setIdle(Integer numberOfNodes) throws RemoteException {

    if (numberOfNodes <= startedNodes) {
        for (int i = 1; i <= numberOfNodes.intValue(); i++) {
            int rank = random.nextInt(aliveNodes.size());
            PastryNode pn = aliveNodes.get(rank);

            Environment env = pn.getEnvironment();
            NodeStateController sc = new NodeStateController();
            env.getSelectorManager().invoke(sc);

            idleNodes.add(pn);
            idleThreads.add(sc);
            aliveNodes.remove(rank);
        }
    }
    else {
        System.out.println("Cannot make idle " +
                numberOfNodes +
                " nodes (number of started nodes = " +
                startedNodes +
                ").");
    }
}

public void killAll() throws RemoteException {
    for (int i = 0; i < aliveNodes.size(); i++) {
        PastryNode pn = aliveNodes.get(i);
        pn.destroy();
    }

    for (int i = 0; i < idleNodes.size(); i++) {
        PastryNode pn = idleNodes.get(i);
        pn.destroy();
    }
}

public DeltaMeasurement getMeasurement (
        String metricName, String measurementId,
        String component, String correlator)
        throws RemoteException {

    P2PMeasurement results = new P2PMeasurement();

    if (metricName.equals("UNIQUE_ENTRIES")) {
        for (int i = 0; i < aliveNodes.size(); i++) {
            PastryNode pn = aliveNodes.get(i);
            results.addMeasurementUnit(
                    pn.getNodeId().toString(),
                    "ALIVE",
                    new Integer(
                    this.componentSwitch(pn, component)).toString());
        }

        for (int i = 0; i < idleNodes.size(); i++) {
            PastryNode pn = aliveNodes.get(i);
```

```
                results.addMeasurementUnit(
                        pn.getNodeId().toString(),
                        "IDLE",
                        new Integer(
                        this.componentSwitch(pn, component)).toString());
            }
        }
        else {
            System.err.println("Unknown metric: " +
                    metricName +
                    ". Returning an empty result.");
        }

        return results;
    }

    private int componentSwitch(PastryNode pn, String component) {
        if (component.equals("ROUTING_TABLE")) {
            return pn.getRoutingTable().numUniqueEntries();
        }
        else if (component.equals("LEAF_SET")) {
            return pn.getLeafSet().getUniqueCount();
        }
        else {
            System.err.println("Unknown component: " +
                    component + ". Returning 0.");
            return 0;
        }
    }

    private class NodeStateController implements Runnable {
        public void run() {
            this.setIdle();
        }

        public synchronized void setIdle() {
            try {
                this.wait();
            }
            catch (InterruptedException e) {
                System.err.println("Cannot make Pastry node idle (" +
                        e.getMessage() + ").");
            }
        }

        public synchronized void setAlive() {
            this.notifyAll();
        }
    }
}
```

3.3 OpenChord

In this section, we show how we plugged DELTA with OpenChord [7] Java-based implementation of the Chord DHT [2]. As for the Pastry use case, we designed a class called `ChordServer` which implements the `P2PRemoteInstance` interface. One can note that since OpenChord does not support any life-cycle control, (1) the state of all nodes is always set to alive and (2) methods related to life-cycle changes are not implemented. The full code of the `ChordServer` class is given in listing 18, and again, the implementation of this class should be considered as an example but not as a reference since althrough it enables us to perform our tests appropriatly, it may have to be reconsidered in order to be used in another context.

Listing 18: The ChordServer class

```
package fr.utt.era.deltaappls.chord;

import java.net.*;
import java.rmi.RemoteException;
import java.util.*;
import java.util.regex.*;

import de.uniba.wiai.lspi.chord.data.URL;
import de.uniba.wiai.lspi.chord.service.*;
import de.uniba.wiai.lspi.chord.service.impl.ChordImpl;

import fr.utt.era.delta.DeltaMeasurement;
import fr.utt.era.delta.p2p.P2PRemoteInstance;

public class ChordServer implements P2PRemoteInstance {

    private String bootstrapAddress;
    private int bootstrapPort;

    private final int FIRST_NODE_PORT = 20000;
    private int NODE_PORT = FIRST_NODE_PORT;

    private Vector<URL> aliveNodes;
    private Vector<ChordImpl> chordNodes;

    public ChordServer() {
        PropertiesLoader.loadPropertyFile();
        aliveNodes = new Vector<URL>();
        chordNodes = new Vector<ChordImpl>();
        bootstrapPort = FIRST_NODE_PORT;
    }

    public DeltaMeasurement getMeasurement (
            String metricName, String measurementId,
            String component, String correlator)
            throws RemoteException {
```

[7]http://sourceforge.net/projects/open-chord/

```
    P2PMeasurement chordMeasurement = new P2PMeasurement();
    int items;
    if (component.equals("FINGER_TABLE")) {
    if (metricName.equals("UNIQUE_ENTRIES")) {

        for (ChordImpl ch : chordNodes) {
            URL peerUrl = ch.getURL();
            String fingerTable = ch.printFingerTable();
            if (fingerTable == null) {
                items = 0;
            } else {
                items = stringOccur(fingerTable, "://");
            }
            String uniqueEntries = new Integer(items).toString();
            chordMeasurement.addMeasurementUnit(
                    peerUrl.toString(), "ALIVE", uniqueEntries);
        }
    } else {
        System.err.println("Unknown metric: " + metricName);
    }
    } else {
        System.err.println("Unknown component: " + component);
    }
    return chordMeasurement;
}

public void setBootstrapAddress(String bootstrapAddress)
        throws RemoteException {
    this.bootstrapAddress = "ocsocket://" + bootstrapAddress + "/";
}

public void createNode() throws RemoteException {

    String bAddr = this.getBootstrapAddress();
    String cAddr = "ocsocket://" +
                   this.getLocalCanonicalHostName() +
                   ":20000/";

    if ((this.NODE_PORT == this.FIRST_NODE_PORT) &&
            (bAddr.equals(cAddr))) {
        this.createOpenChordOverlayNetwork(
                this.getBootstrapAddress());
        this.NODE_PORT += 1;
    } else {
        this.createOpenChordPeer(this.getNextNodeAddress());
    }
}

private void createOpenChordOverlayNetwork(String peerAddress) {

    try {
        ChordImpl chord = new ChordImpl();
        URL peerUrl = new URL(peerAddress);
        chord.create(peerUrl);

```

```
            aliveNodes.addElement(peerUrl);
            chordNodes.addElement(chord);

        } catch (MalformedURLException e) {
            throw new RuntimeException(e);
        } catch (ServiceException e) {
            throw new RuntimeException("Could not create DHT ! ", e);
        }
    }

    private void createOpenChordPeer(String peerAddress) {
        try {
            ChordImpl chord = new ChordImpl();
            URL peerUrl = new URL(peerAddress);
            chord.join(peerUrl, new URL(getBootstrapAddress()));

            aliveNodes.addElement(peerUrl);
            chordNodes.addElement(chord);

        } catch (MalformedURLException e) {
            throw new RuntimeException(e);
        } catch (ServiceException e) {
            throw new RuntimeException("Could not join DHT ! ", e);
        }
    }

    public void kill(Integer numberOfNodes) throws RemoteException {
        Random random = new Random();
        Chord chord = new ChordImpl();
        if (numberOfNodes <= aliveNodes.size()) {
            try {

                for (int i = 1; i <= numberOfNodes; i++) {
                    int rank = random.nextInt(aliveNodes.size());
                    URL peerUrl = aliveNodes.get(rank);
                    chord.setURL(peerUrl);
                    chord.leave();
                    aliveNodes.remove(rank);
                }
            } catch (ServiceException e) {
                e.printStackTrace();
            }
        } else {
            System.err.println("Cannot kill " + numberOfNodes +
                    " nodes (number of started nodes = " +
                    aliveNodes.size() + ").");
        }
    }

    public void killAll() throws RemoteException {
        Chord chord = new ChordImpl();
        try {
            for (URL peerUrl : aliveNodes) {
                chord.setURL(peerUrl);
                chord.leave();
            }
        } catch (ServiceException e) {
            e.printStackTrace();
        }
        aliveNodes.clear();
    }

    public void setAlive() throws RemoteException {
        // This method is not used with our OpendChord connector.
    }

    public void setIdle(Integer arg0) throws RemoteException {
        // This method is not used with our OpendChord connector.
    }

    private String getBootstrapAddress() {
        return bootstrapAddress;
    }

    private String getNextNodeAddress() {
        String result = URL.KNOWN_PROTOCOLS.get(URL.SOCKET_PROTOCOL) +
                "://" + getLocalCanonicalHostName() +
                ":" + NODE_PORT + "/";
        return result;
    }

    private String getLocalCanonicalHostName() {
        String result = null;
        try {
            result = InetAddress.getLocalHost().getCanonicalHostName();
        } catch (UnknownHostException e) {
            e.printStackTrace();
        }
        return result;
    }

    private int stringOccur(String text, String string) {
        return regexOccur(text, Pattern.quote(string));
    }

    private int regexOccur(String text, String regex) {
        Matcher matcher = Pattern.compile(regex).matcher(text);
        int occurence = 0;
        while (matcher.find()) {
            occurence++;
        }
        return occurence;
    }
}
```

4 Appendix

4.1 Full version of the point scenario

Listing 19: Full version of the point scenario

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE scenario SYSTEM 'scenario.dtd'>
<scenario>
    <metadata>
        <title>Points: scenario 1</title>
        <version>version 1 : 12/10/2007</version>
        <authors>Marc LEMERCIER</authors>
        <description>Affiche Points</description>
    </metadata>
    <topology>
        <slave name='slave1' hostname='slave1.utt.fr' />
        <slave name='slave2' />
        <slave name='slave3' />
    </topology>
    <tasks>
        <task ID='1' name='CreatePoints'>
            <time>
                <start>1</start>
                <stop>2</stop>
                <occurrence>2</occurrence>
            </time>
                <targets select='list'>
                <slave name='slave1' />
                <slave name='slave2' />
            </targets>
            <action mode='random'>
                <method-name>createPoint</method-name>
                <method-params>
                    <param>
                        <param-type>java.lang.String</param-type>
                        <param-name>label</param-name>
                        <param-value>point</param-value>
                    </param>
                    <param>
                        <param-type>java.lang.Integer</param-type>
                        <param-name>x</param-name>
                        <param-value>10</param-value>
                    </param>
                    <param>
                        <param-type>java.lang.Integer</param-type>
                        <param-name>y</param-name>
                        <param-value>20</param-value>
                    </param>
                </method-params>
            </action>
        </task>
        <task ID='2' name='measurement'>
            <time>
                <start>5</start>
                <stop>9</stop>
                <occurrence>2</occurrence>
            </time>
            <targets select='topology' />
            <action mode='all'>
                <method-name>getMeasurement</method-name>
                 <method-params>
                     <param>
                         <param-type>java.lang.String</param-type>
                         <param-name>metricName</param-name>
                         <param-value>DISTANCE</param-value>
                     </param>
                     <param>
                         <param-type>java.lang.String</param-type>
                         <param-name>measurementId</param-name>
                         <param-value>0x0001</param-value>
                     </param>
                     <param>
                         <param-type>java.lang.String</param-type>
                         <param-name>component</param-name>
                         <param-value>null</param-value>
                     </param>
                     <param>
                         <param-type>java.lang.String</param-type>
                         <param-name>correlator</param-name>
                         <param-value>null</param-value>
                     </param>
                 </method-params>
            </action>
        </task>
        <task ID='2' name='measurement'>
            <time>
                 <start>5</start>
                 <stop>9</stop>
                 <occurrence>2</occurrence>
            </time>
            <targets select='topology' />
            <action mode='all'>
                 <method-name>getMeasurement</method-name>
                 <method-params>
                     <param>
                         <param-type>java.lang.String</param-type>
                         <param-name>metricName</param-name>
                         <param-value>DISTANCE</param-value>
                     </param>
                     <param>
                         <param-type>java.lang.String</param-type>
                         <param-name>measurementId</param-name>
                         <param-value>0x0001</param-value>
                     </param>
                     <param>
                         <param-type>java.lang.String</param-type>
                         <param-name>component</param-name>
                         <param-value>null</param-value>
                     </param>
                     <param>
                         <param-type>java.lang.String</param-type>
                         <param-name>correlator</param-name>
                         <param-value>null</param-value>
                     </param>
                 </method-params>
            </action>
        </task>
    </tasks>
</scenario>
```

4.2 scenario DTD

Listing 20: DTD of a scenario

```
<?xml version='1.0' encoding='UTF-8'?>
<!ELEMENT scenario (tasks|topology|metadata)*>
<!ELEMENT metadata (abstract|authors|version|title)*>
<!ELEMENT title (#PCDATA)>
<!ELEMENT version (#PCDATA)>
<!ELEMENT authors (#PCDATA)>
<!ELEMENT abstract (#PCDATA)>
<!ELEMENT topology (slave)*>
<!ELEMENT slave EMPTY>
<!ATTLIST slave
   hostname CDATA #IMPLIED
   name CDATA #REQUIRED
>
<!ELEMENT tasks (task)*>
<!ELEMENT task (action|targets|time)*>
<!ATTLIST task
   name CDATA #IMPLIED
   ID CDATA #IMPLIED
>
<!ELEMENT time (occurrence|stop|start)*>
<!ELEMENT start (#PCDATA)>
<!ELEMENT stop (#PCDATA)>
<!ELEMENT occurrence (#PCDATA)>
<!ELEMENT targets (slave)*>
<!ATTLIST targets
   select (list|topology) #REQUIRED
>
<!ELEMENT action (method-params|method-name)*>
<!ATTLIST action
   mode CDATA #IMPLIED
>
<!ELEMENT method-name (#PCDATA)>
<!ELEMENT method-params (param)*>
<!ELEMENT param (param-value|param-name|param-type)*>
<!ELEMENT param-type (#PCDATA)>
<!ELEMENT param-name (#PCDATA)>
<!ELEMENT param-value (#PCDATA)>
```

References

[1] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings*

of the IFIP/ACM International Conference on Distributed Systems Platforms - Middleware'01, number 2218 in LNCS, pages 329–350. Springer-Verlag, 2001.

[2] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures and Protocols for Computer Communication - SIGCOMM'01*, pages 149–160. ACM Press, 2001.